****

编译原理实验报告二

语法分析器的实现

姓名： 井劭杰

学号： 71116234

东南大学计算机科学与工程学院、软件学院

School of Computer Science & Engineering

College of Software Engineering

Southeast University

二0 一 九 年 一 月

# 实验目的

对于输入的LL(1)文法，构造其PPT，并对输入串分析其是否为该文法的句子。

# 实验内容

##### 1、Input

Stream of characters

CFG(Combination of CFGs of some classes of sentences)

##### 2、Output

Sequence of derivations if top-down syntax analyzing methods are used.

Sequence of reductions if bottom-up syntax analyzing methods are used.

##### Classes of words are defined by yourself

##### Error handling may be included

# 实验方法

##### 总体思想

a)Construct LL(1) parsing table based on the CFG

b)Design the program using LL(1) paring table

**（注：实验中全部使用$代替ε，使用#代替$R）**

##### **求FIRST集**

如果产生式右部第一个字符为终结符，则将其计入左部的FIRST集

如果产生式右部第一个字符为非终结符

->求该非终结符的FIRST集

->将该非终结符的非$的FIRST集计入左部的FIRST集

->若存在$，则将指向产生式的指针右移

->若不存在$，则停止遍历该产生式，进入下一个产生式

->若已经到达产生式的最右部的非终结符，则将$加入左部的FIRST集

处理FIRST集中重复的终结符

##### **求FOLLOW集**

对于文法G中每个非终结符A构造FOLLOW(A)，连续使用下面的规则，直到得到的FOLLOW集不再有扩充，即为最终的FOLLOW集

(1) 对于文法的开始符号S，置#于FOLLOW(S)中;

(2) 若A->aBb是一个产生式，则把非$的FIRST(b)加至 FOLLOW(B)中;

(3) 若A->aB是一个产生式，或A->aBb是一个产生式而b=>$(即$∈FIRST(b))则将FOLLOW(A)加至FOLLOW(B)中

##### **构造PPT**

对文法G的每个产生式A->a

->对每个终结符a∈FIRST(a)，把A->a加至[A,a]中;

->若$∈FIRST(a)，则对任何b∈FOLLOW(A)把A->a加至[A,b]中;

在PPT中的空处标注error

##### **分析输入符号串**

对于STACK栈顶符号X和当前读头指向的符号a，程序根据以下三种情况进行分析操作：

(1) 若 X=a=#,则accept，停止分析

(2) 若 X=a≠#，则把X出栈,让读头右移一位

(3) 若 X 是一个非终结符,则查看PPT：若[X,a]中有产生式,那么首先把X出栈，然后把产生式的右部进栈(若右部为 ε,则不进栈)，再次进行分析操作；若[A,a]中为error，则输入符号串不符合该文法

# 实验代码

Pre类负责求出FIRST和FOLLOW，TableStack类负责构造PPT以及输入字符串的分析

#include "stdafx.h"

#include <iostream>

#include <iomanip>

#include <string>

#include <cstring>

#include <cmath>

#include <stack>

#include <vector>

#include <string>

#include <set>

#include <algorithm>

#include <map>

#define MAX 100

using namespace std;

struct production //产生式的结构

{

char left;

string right;

};

class Pre

{

protected:

int T;

production analysis[MAX]; //输入文法分析

set<char> FIRST[MAX];//First集

set<char> FOLLOW[MAX];//Follow集

vector<char> terminal\_NoEmpty; //去$终结符

vector<char> terminal;//终结符

vector<char> nonterminal;//非终结符

public:

Pre() :T(0) {}

//获得在终结符集合中的下标

bool isNotSymbol(char c)

{

if (c >= 'A' && c <= 'Z')

return true;

return false;

}

int get\_index(char set)

{

for (int i = 0; i < nonterminal.size(); i++) {

if (set == nonterminal[i])

return i;

}

return -1;

}

//获得在非终结符集合中的下标

int get\_nindex(char set)

{

for (int i = 0; i < terminal\_NoEmpty.size(); i++) {

if (set == terminal\_NoEmpty[i])

return i;

}

return -1;

}

//得到First集合

void get\_first(char Set)

{

int flag = 0;

int tag = 0;

for (int i = 0; i < T; i++) {

if (analysis[i].left == Set) {//产生式左部匹配

if (!isNotSymbol(analysis[i].right[0])) {//终结符直接加入First

FIRST[get\_index(Set)].insert(analysis[i].right[0]);

}

else {

for (int j = 0; j < analysis[i].right.length(); j++) {

if (!isNotSymbol(analysis[i].right[j])) {//终结符直接加入First

FIRST[get\_index(Set)].insert(analysis[i].right[j]);

break;

}

//cout << analysis[i].right[j] << endl; //输出测试用

get\_first(analysis[i].right[j]);

set<char>::iterator temp;

for (temp = FIRST[get\_index(analysis[i].right[j])].begin();

temp != FIRST[get\_index(analysis[i].right[j])].end(); temp++) {

if (\*temp == '$')

flag = 1;

else

FIRST[get\_index(Set)].insert(\*temp); //First(Y)中的非$终结符加入First(X)

}

if (flag == 0)

break;

else {

tag += flag;

flag = 0;

}

}

if (tag == analysis[i].right.length())

FIRST[get\_index(Set)].insert('$'); //所有右部First(Y)都有$，将$加入First

}

}

}

}

//得到Follow集合

void get\_follow(char Set)

{

for (int i = 0; i < T; i++) {

int index = -1;

int length = analysis[i].right.length();

for (int j = 0; j < length; j++) {

if (analysis[i].right[j] == Set) { //找出该产生式下标

index = j;

break;

}

}

if (index != -1 && index < length - 1) {

char next = analysis[i].right[index + 1];

if (!isNotSymbol(next))

FOLLOW[get\_index(Set)].insert(next);

else {

int temp = 0;

set<char>::iterator it;

for (it = FIRST[get\_index(next)].begin(); it != FIRST[get\_index(next)].end(); it++) {

if (\*it == '$')

temp = 1;

else

FOLLOW[get\_index(Set)].insert(\*it);

}

if (temp && analysis[i].left != Set) {

get\_follow(analysis[i].left);//递归

char ch = analysis[i].left;

set<char>::iterator it;

for (it = FOLLOW[get\_index(ch)].begin(); it != FOLLOW[get\_index(ch)].end(); it++) {

FOLLOW[get\_index(Set)].insert(\*it);

}

}

}

}

else if (index != -1 && index == length - 1 && Set != analysis[i].left) {

get\_follow(analysis[i].left);//递归

char ch = analysis[i].left;

set<char>::iterator it;

for (it = FOLLOW[get\_index(ch)].begin(); it != FOLLOW[get\_index(ch)].end(); it++) {

FOLLOW[get\_index(Set)].insert(\*it);

}

}

}

}

//处理得到First和Follow集合

void get\_result()

{

cout << endl;

cout << endl;

cout << "YOU CAN ONLY USE THE PREPROCESSED-PRODUCTIONS!!!" << endl;

cout << endl;

cout << endl;

cout << "Please input the number of productions: ";

cin >> T;

string s;

cout << "enter the productions(use $ to replace ε): " << endl;

for (int temp = 0; temp < T; temp++) {

cin >> s;

string t = "";

for (int i = 0; i < s.length(); i++) {

if (s[i] != ' ')

t += s[i];

}

analysis[temp].left = t[0];

for (int j = 3; j < t.length(); j++)

analysis[temp].right += t[j];

for (int j = 0; j < t.length(); j++) {

if (t[j] != '-'&&t[j] != '>') {

if (isNotSymbol(t[j])) {

int flag = 0;

for (int a = 0; a < nonterminal.size(); a++) {

if (nonterminal[a] == t[j]) {

flag = 1;

break;

}

}

if (!flag)

nonterminal.push\_back(t[j]);

}

else {

int flag = 0;

for (int a = 0; a < terminal.size(); a++) {

if (terminal[a] == t[j])

{

flag = 1;

break;

}

}

if (!flag)

terminal.push\_back(t[j]);

}

}

}

}

terminal.push\_back('#');

for (int i = 0; i < nonterminal.size(); i++)

{

get\_first(nonterminal[i]);

}

for (int i = 0; i < nonterminal.size(); i++)

{

if (i == 0)

FOLLOW[0].insert('#');

get\_follow(nonterminal[i]);

}

for (int i = 0; i < terminal.size(); i++)

{

if (terminal[i] != '$')

terminal\_NoEmpty.push\_back(terminal[i]);

}

}

//输出First和Follow集合

void displayFF()

{

cout << "FIRST集合为" << endl;

for (int i = 0; i < nonterminal.size(); i++)

{

cout << nonterminal[i] << ": ";

set<char>::iterator it;

for (it = FIRST[i].begin(); it != FIRST[i].end(); it++)

cout << \*it << " ";

cout << endl;

}

cout << "FOLLOW集合为" << endl;

for (int i = 0; i < nonterminal.size(); i++)

{

cout << nonterminal[i] << ": ";

set<char>::iterator it;

for (it = FOLLOW[i].begin(); it != FOLLOW[i].end(); it++)

cout << \*it << " ";

cout << endl;

}

}

};

class TableStack :public Pre

{

protected:

vector<char> analysis\_stack; //分析栈

vector<char> left\_analysis;//剩余输入串

int PPT[100][100];//预测表

public:

TableStack() {

memset(PPT, -1, sizeof(PPT));

}

//得到预测表

void get\_PPT()

{

for (int i = 0; i < T; i++)

{

char ch = analysis[i].right[0];

if (!isNotSymbol(ch))

{

if (ch != '$')

PPT[get\_index(analysis[i].left)][get\_nindex(ch)] = i;

if (ch == '$')

{

set<char>::iterator it;

for (it = FOLLOW[get\_index(analysis[i].left)].begin(); it != FOLLOW[get\_index(analysis[i].left)].end(); it++)

{

PPT[get\_index(analysis[i].left)][get\_nindex(\*it)] = i;

}

}

}

else

{

set<char>::iterator it;

for (it = FIRST[get\_index(ch)].begin(); it != FIRST[get\_index(ch)].end(); it++)

{

PPT[get\_index(analysis[i].left)][get\_nindex(\*it)] = i;

}

if (FIRST[get\_index(ch)].count('$') != 0)

{

set<char>::iterator s;

for (s = FOLLOW[get\_index(analysis[i].left)].begin(); s != FOLLOW[get\_index(analysis[i].left)].end(); s++)

{

PPT[get\_index(analysis[i].left)][get\_nindex(\*s)] = i;

}

}

}

}

}

//分析栈的处理

void analyExp(string s)

{

for (int i = s.length() - 1; i >= 0; i--)

left\_analysis.push\_back(s[i]);

analysis\_stack.push\_back('#');

analysis\_stack.push\_back(nonterminal[0]);

while (left\_analysis.size() > 0)

{

//分析栈

string outs = "";

for (int i = 0; i < analysis\_stack.size(); i++)

outs += analysis\_stack[i];

cout << setw(15) << outs;

//剩余输入串

outs = "";

for (int i = left\_analysis.size() - 1; i >= 0; i--)

outs += left\_analysis[i];

cout << setw(15) << outs;

char char1 = analysis\_stack[analysis\_stack.size() - 1];

char char2 = left\_analysis[left\_analysis.size() - 1];

if (char1 == char2 && char1 == '#') {

cout << setw(15) << "Accepted!" << endl;

return;

}

if (char1 == char2) {

analysis\_stack.pop\_back();

left\_analysis.pop\_back();

cout << setw(15) << char1 << "match " << endl;

}

else if (PPT[get\_index(char1)][get\_nindex(char2)] != -1) {

int temp = PPT[get\_index(char1)][get\_nindex(char2)];

analysis\_stack.pop\_back();

if (analysis[temp].right != "$") {

for (int i = analysis[temp].right.length() - 1; i >= 0; i--)

analysis\_stack.push\_back(analysis[temp].right[i]);

}

cout << setw(15) << analysis[temp].right << endl;

}

else {

cout << setw(15) << "Error!" << endl;

return;

}

}

}

//输出

void print()

{

for (int i = 0; i < terminal\_NoEmpty.size(); i++)

{

cout << setw(10) << terminal\_NoEmpty[i];

}

cout << endl;

for (int i = 0; i < nonterminal.size(); i++)

{

cout << nonterminal[i] << ": ";

for (int j = 0; j < terminal\_NoEmpty.size(); j++)

{

if (PPT[i][j] == -1)

cout << setw(10) << "error";

else

cout << setw(10) << analysis[PPT[i][j]].right;

}

cout << endl;

}

}

//结合处理

void getAns()

{

get\_result();

displayFF();

get\_PPT();

print();

string ss;

cout << "请输入符号串（#代表$R）：" << endl;

cin >> ss;

cout << setw(15) << "分析栈" << setw(15) << "剩余输入串" << setw(15) << "推导式" << endl;

analyExp(ss);

}

};

int main()

{

TableStack t;

t.getAns();

system("pause");

return 0;

}

# 结果展示

**输入文法（$代表ε）：**

E->TA

A->+TA

A->$

T->FB

B->\*FB

B->$

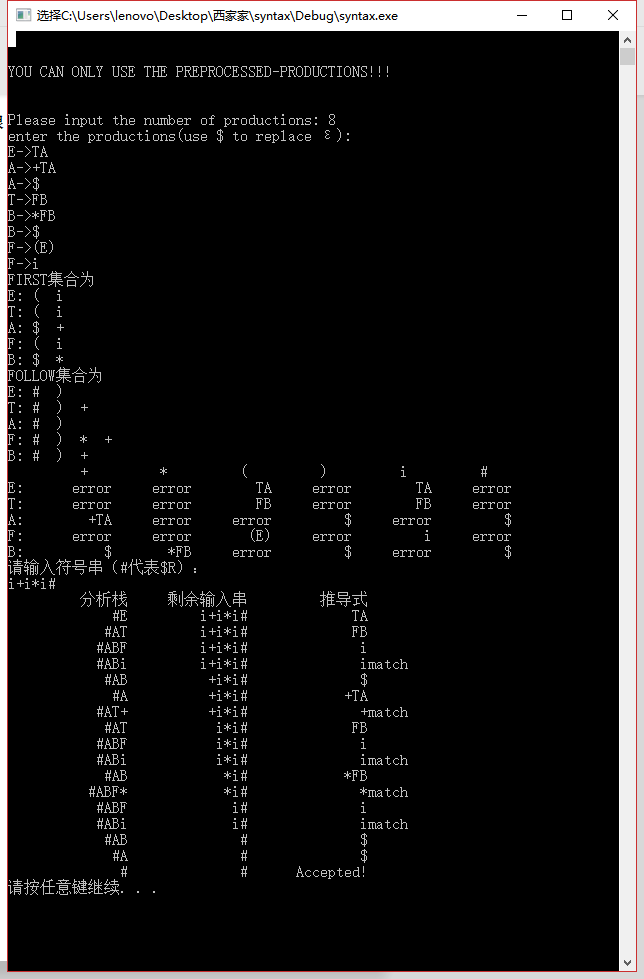
F->(E)

F->i

**输入字符串（#代表$R）：**

i + i \* i #

**程序运行结果：**



# 心得体会

本次试验的关键是求出FIRST和FOLLOW，之后的构造PPT和根据PPT分析输入字符串都比较简单。通过本次实验，我更加深刻地理解了LL(1)分析法的分析过程和不足，也认识到了编译中的文法是多变的，但是编译的原理和思想是统一的，只要把握住编译原理的核心思想就能灵活地处理各种情况，而不应该只会死板地套公式。